J Internet Serv Appl (2012) 3:15–22
DOI 10.1007/s13174-011-0048-5

SI: FOME - THE FUTURE OF MIDDLEWARE

Developing highly complex distributed systems: a software engineering perspective

Marco Autili · Paola Inverardi · Patrizio Pelliccione · Massimo Tivoli

Received: 8 November 2011 / Accepted: 12 November 2011 / Published online: 25 November 2011

Abstract What is a highly complex distributed system in the future era? What are the needs that may drive the development of such systems? And what is their life cycle? Is there any new challenge for Software Engineering (SE)? In this paper, we try to provide a partial answer to the above questions by characterizing few application domains that we consider of raising interest in the next years. Our thesis is that there is a need to rethink the whole software process for such systems. The traditional boundaries between static and dynamic activities disappear and development support mingles with run time support thus invading the middleware territory.

Keywords Future Internet · Service-oriented computing · Service-oriented middleware · Cloud computing · Middleware-based software process

1 Introduction

The near future envisions a pervasive heterogeneous computing infrastructure that enables end-user, with different needs and expectations, to provide and access software services on a variety of devices. This vision leads to the development of software systems that are more and more complex and large scale, namely Highly Complex Distributed Systems (HCDSs). To ensure that users experience the best Quality of Service (QoS) according to their needs and specific contexts of use, HCDSs need to be context-aware, adaptable, and dependable. Heterogeneity of the underlying communication infrastructure, mobility inducing changes to the availability of resources and continuously evolving requirements also call for adaptiveness and context-awareness.

The development and the execution of HCDSs is a big challenge and it is far to be solved. Indeed, HCDSs cannot rely on the classical desktop-centric assumption that the system execution environment and the networking environment are known a priori at design-time. Rather, HCDSs will need to cope with variability, as software gets deployed on an increasingly large diversity of computing platforms and operates in different networking environments that cannot be statically anticipated. Therefore, context-awareness and adaptiveness become two key aspects to consider while developing and running HCDSs. While providing/consuming services, HCDSs need to be aware of and adaptive to their *context*, that is, the combination of user-centric data (e.g., information of interest for users according to their current circumstance, like requested QoS) and resource/computer-centric data (e.g., status of devices and network, like availability of resources).

Engineering the development and supporting the execution of HCDSs raises numerous challenges that involve languages, methods, and tools, spanning from design- to run-time, from validation to evolution. Although different software engineering techniques have been studied to deal with distributed, adaptive, and context-aware systems, these initiatives are mainly confined to specific research areas: software architectures [4, 7, 19], middleware [29, 32], component-based development [30], service-oriented sys-

M. Autili · P. Inverardi · P. Pelliccione (✉) · M. Tivoli
Università degli Studi di L'Aquila, L'Aquila, Italy
e-mail: patrizio.pelliccione@di.univaq.it

M. Autili
e-mail: marco.autili@di.univaq.it

P. Inverardi
e-mail: paola.inverardi@di.univaq.it

M. Tivoli
e-mail: massimo.tivoli@di.univaq.it

tems [11], etc. There is consensus, however, that the results obtained so far are inadequate to deal with the challenges that HCDSs will face in the future. HCDSs require a coordinated strategy that covers the whole life cycle within one large ambitious action. As firstly stated in [23] and then mentioned in [3, 6, 24], the software development process life cycle needs to be rethought by breaking the traditional division among development phases. This is achieved by moving some development activities from design-time to deployment- and run-time, hence asking for new and more efficient techniques to support run-time activities. Hence, for HCDSs, the way software will be produced and used radically changes. From an engineering perspective, enabling user-centrism, context-awareness, and adaptiveness requires the adoption of a dynamic development process that never stabilizes, rather it is permanently under maintenance.

Software engineering best practices suggest the exploitation of a middleware that, through the provision of proper features, supports distributed applications by masking the distribution and heterogeneity of the execution and networking environment [25]. In this respect, our thesis is that, for HCDSs, the frontier between application and middleware cannot be a priori fixed once the application domain has been identified, rather it varies as the application concerns vary. This moves forward the view of proposed middleware-based development processes, where an application-domain-specific middleware is selected and employed once and for all, as proposed in [25]. The envisioned dynamic development process includes a perpetual phase in which the run-time-support features, offered by the inherently dynamic middleware-layer, are selected and customized, assembled and evolved according to the characteristics of: the context, the execution environment, and the user needs.

In this paper, we propose a perpetual development process model for HCDSs (Sect. 2). We then consider different application domains together with their supporting middleware to show how for HCDSs the frontier between application- and middleware-layer is unlikely to be tangible (Sect. 3).

2 A perpetual engineering process model for highly complex distributed systems

Our vision is that the engineering of HCDSs needs a dynamic software process where development-time activities mingle with run-time support, hence invading the middleware territory. This means that the dynamic software process we envisage asks for an explicit characterization of the properties of the software artifacts and of their assumptions, by taking into account perpetual evolution of both the application- and middleware-layer.

We recall that HCDSs need to be always in a up-to-date state, that is, they must be able to dynamically evolve in order to interact with a continuously changing environment that might affect the system behavior. The changes might be of different nature, e.g., changes of network topology (e.g., due to mobility), where new nodes appear and existing nodes vanish; changes of the set of available components, that is new components may become available, existing components may disappear, and new connections may be dynamically established; changes in the system requirements that can happen while the system is providing services to its users.

In general, the process view focuses on the set of activities, which characterize the production and the operation of a software system, and the set of artifacts elaborated/produced by such activities. These are traditionally divided into activities related to the actual production of the software system (i.e., before deployment) and activities that are performed when the system can be executed and goes into operation (i.e., after deployment). Specification, Design, Validation, and Evolution activities vary depending on the organization and the type of system being developed. Each activity works on suitable abstractions (models) of the system and requires its language, methods and tools. Therefore, in the process life cycle, systems are represented (through models) at very different levels of abstraction, from requirements specification to code.

So far, software complexity has been addressed by exacerbating the dichotomy development-/static-/compile-time versus execution-/dynamic-/interpret-time concentrating the management of and reasoning on models, as much as possible, at development-time. The mobile, ubiquitous, and heterogeneous scenario in which HCDSs operate poses new requirements on this standard process. The evolutionary nature (adaptiveness, context-awareness, user-centrism) of HCDSs makes unfeasible a static-oriented development process since it would require, before the system is in execution, to predict the system behavior with respect to, virtually, any possible change, from application- down to middleware-layer. In the scenario in which HCDSs operate, possibly evolving user requirements and needs have to be perpetually guaranteed through adaptation. Whatever the change nature is, when it occurs, the current state of the system, including its current context and execution environment, is observed in order to suitably react to the change. This means that the needed models must be available at run-time so that reasoning steps on them become part of the adaptation activity carried on while the system is executing, i.e., at run-time. Such a *perpetual engineering process model* therefore has to explicitly account for complex reasoning steps at run-time when all the necessary pieces of information of the system are available.

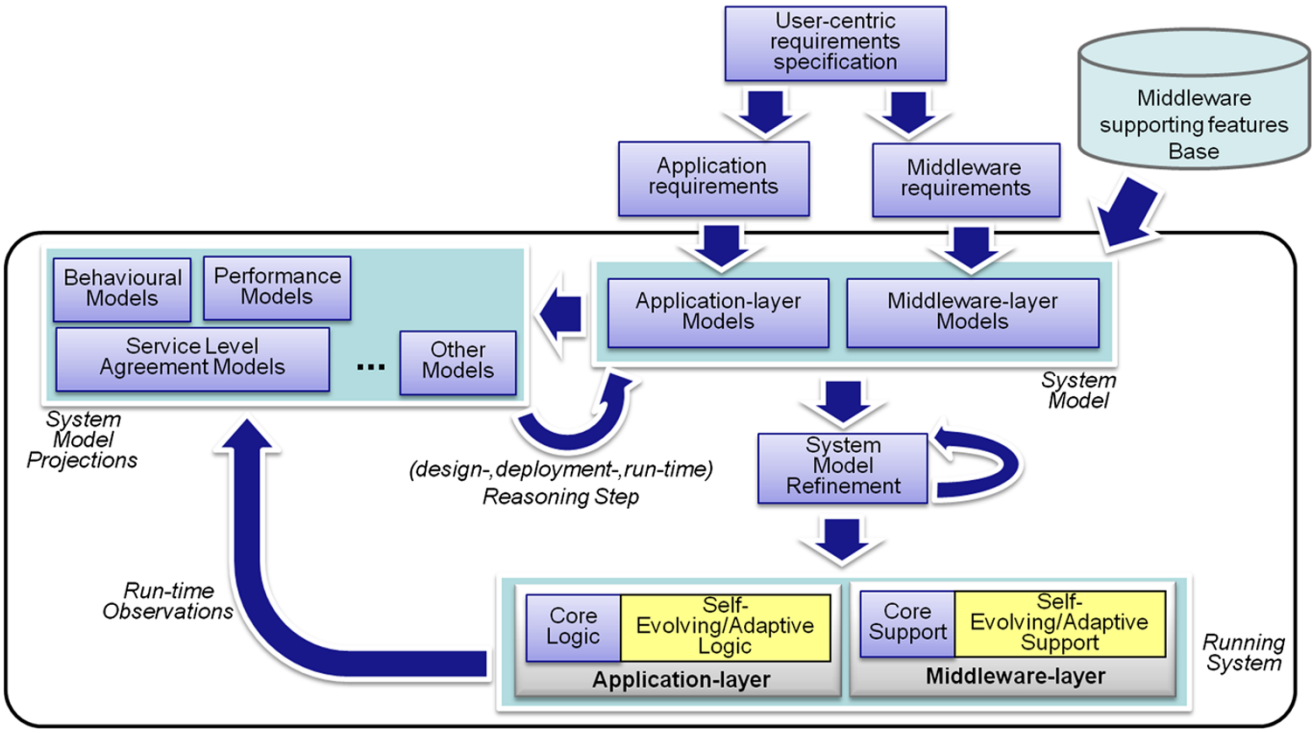

Fig. 1 The perpetual engineering process model

The envisioned process model sketched in Fig. 1 intends to support the systematic development of HCDSs from their design to their execution. The process model is based on model transformation techniques that serve the role of (i) refining user-centric requirement specifications to produce system code by accounting for middleware concerns, (ii) producing reasoning-technique-specific models (i.e., system model projections) from system models, embedding reasoning feedbacks into system models, and keeping them synchronized, (iii) propagating the interpretation of run-time observations of the running system to reasoning-technique-specific models, which in turn will enact possible changes to system models.

Referring to the point (i) above, as shown in Fig. 1, system models are obtained by refining user-centric requirement specifications into application and middleware requirements, and then into system models. System models are the starting point for a refinement chain that leads to the code. This approach promotes system's code synthesis that starts from the (a) application-layer models of the system and from (b) the middleware-layer models. Input (a) accounts for functional and nonfunctional concerns. Input (b) comes from the exploitation of a *middleware-layer supporting features* base. Indeed, our process model envisions a scenario in which middleware-layer supporting features are publicly available (e.g., into a registry) and can be discovered and selected to satisfy the specified middleware-layer requirements. Then model-to-code transformations are used to build both the core code and the "generic code" of the system, from the application- down to the middleware-layer. The core code is the frozen unchanging portion of the system's code. The generic code is an adaptive code that embodies a certain degree of variability (i.e., different ways of implementing a subsystem) making it capable to evolve. This code portion is evolving in the sense that, based on contextual information and possible changes of the user needs, the variability can be solved by considering alternative code.

As discussed in point (ii), model-driven techniques support the integration of different reasoning techniques, by propagating reasoning results into the various models, and by keeping them synchronized. Indeed, changes occurring in a model may have a strong impact on other interoperating models. Therefore, through changes propagation, models are kept in a consistent state. This issue becomes a complex task when dealing with multiple model notations. such a task is inevitable and requires to be managed by a dedicated approach. Recent development of model-driven techniques to bridge among different notations [17] makes this vision feasible.

Finally, referring to the point (iii), the interpretation of run-time observations needs to be propagated back to the system models level. These observations are used to feed the reasoning process that can induce changes to the system model projections, as well as to the system models [8, 16].

3 Breaking the frontier between application- and middleware-layer

Middleware abstract the networking and computing environment by providing well-known reusable solutions to recurrent problems like heterogeneity, interoperability, security, and dependability. In this respect, since all these dimensions match distributed application requirements, middleware facilitate the development of distributed applications.

Depending on the application domain of interest, different supporting middleware have been developed. As far as evolution of HCDSs is concerned, in the following we consider some application domains together with their computing paradigms. We then discuss how from software engineering perspective the frontier between application- and middleware-layer is unlikely to be distinguishable, once and for all, prior deployment.

3.1 Context-aware adaptable systems

During the last decade, context-awareness and adaptation have been receiving significant attention in many research areas [2, 13, 21, 22]. The need for adapting software applications becomes obvious for new development paradigms, such as mobile and pervasive computing, when dealing with context variations. The main drivers of the pervasive computing paradigm are then context-aware and adaptable software applications for resource-constrained mobile devices. They are characterized by their heterogeneity (e.g., in terms of hardware platform, operating system, and programming language) and limitedness (e.g., limited processor speed, limited battery lifetime, and slow unreliable and intermittent connection). Context awareness refers to the capability of observing pieces of information of the surrounding environment of the system that may influence the system behavior, and that are out of the system control (e.g., networks, user needs, resources, device status). Adaptability refers to systems whose behavior can be adjusted in response to the context variations in order to keep requirements satisfied.

The development and execution of context-aware adaptable applications are big challenges for the research community. The main difficulty is to provide (i) an easy-to-use and powerful programming technique for developers to actually program adaptable applications, and (ii) a context-aware run-time support to properly handle contextual situations. In the literature, many valuable approaches have been proposed to this purpose, e.g., [15, 18, 20, 21, 27, 28, 34–37]. In particular, the work in [21] points out that current mainstream programming languages and runtime environments provide little help for specifying adaptation and for supporting context awareness. This leads to a system design that is more complex and convoluted than needed; this motivates the authors to discuss the need for a context-oriented programming approach able to support context-dependent variations.

In general, three different development approaches toward adaptable applications can be distinguished: (i) self-contained applications that embed the adaptation logic as a part of the application itself; therefore, they are a priori instructed on how to handle dynamic changes in the environment; (ii) tailored applications that are the result of an adaptation process which has been applied on a generic version of the application, at deployment time at latest; (iii) middleware-based adaptable applications in which the middleware embeds the adaptation logic in the form of metalevel information on how the applications can be adapted.

Self-contained adaptable applications are inherently dynamic but suffer the pay-off of the inevitable overhead imposed by the adaptation logic. On the contrary, tailored adapted applications are suitable also for limited devices, but are dynamic only with respect to the environment at deployment time, while remaining static with respect to the actual execution, i.e., they cannot natively self-adapt to runtime changes in the execution environment. Middleware-based adaptable applications emerge as the result of a line of research, started in early 2000, looking for general purposes solutions to externalize adaptation mechanisms from the application logic implementation [12]. To enable such an externalization, many middleware-based solutions have been proposed, e.g, [5, 10]. While externalizing adaptation mechanisms, middleware not only virtualize communications but also offer proper supporting features to access and manage the context. However, when tightly coupled to mobile devices and ubiquitous computing, context-aware adaptable applications become distributed in nature and intrinsically complex to be engineered and implemented. Engineering middleware-based context-aware adaptable applications for mobile devices is not easy due the limitedness and heterogeneity of the targeted execution environment. Indeed, it is unfeasible a “one-for-all” and “all-in-one” middleware solution that embeds the adaptation logic together with all the needed metalevel information, while fitting the resource constraints imposed by the different mobile devices.

A right trade-off can be obtained by a hybrid approach that brings together the advantages of middleware-based solutions, tailored and self-contained solutions. The approach would allow for externalizing adaptation by customizing/selecting not only the application logic but also the middleware features in support of adaptation mechanisms. Self-containment would bring the possibility of embedding those portions of the adaptation logic that are application specific and, as such, cannot be offered by the middleware as general reusable solutions to frequently encountered problems. Clearly, such a mixed approach requires a future engineering process where the core logic of the application and the core support of the middleware are both subject to evolution. That is, after a certain tailoring step (that takes into

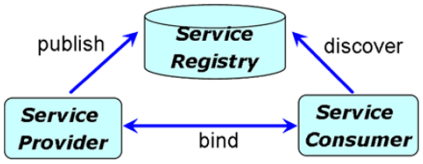

Fig. 2 Service oriented interaction pattern

account the resources status of the devices, its context, and user needs in a given instant), the core logic and the core support may evolve in response to context changes, e.g., some resources degrade or disappear.

3.2 Service-oriented systems

The notion of Service-Oriented Architecture (SOA) has been receiving significant attention within the software engineering community. SOA has been largely accepted as a well founded reference architectural style for a wide class of distributed software systems. SOA relies on the Service-Oriented Computing (SOC) paradigm that considers services as building blocks for developing distributed applications. Networked devices and their hosted applications are then abstracted as autonomous loosely coupled services that are amenable to be integrated into a network of services to create flexible and dynamic processes, thanks to the service-oriented interaction pattern (depicted in Fig. 2). With network connectivity being embedded in most computing devices, any networked device may seamlessly consume but also provide software applications over the network. SOC then introduces natural design abstractions to deal with ubiquitous networking environments [9].

Service-Oriented Middleware (SOM) supports the service-oriented interaction pattern through the provision of proper features for deploying, publishing, discovering, and binding services at run time, as well as, features for supporting extra-functional crosscutting concerns such as service semantics and QoS. Services play a central role in this vision as effective means to achieve interoperability between heterogeneous parties of a business process and independence from the underlying infrastructure. At the same time, they offer an open platform to build new value added service-based systems as a collaboration of available services. To support this view, the notion of *choreography* is recently receiving growing interest as a key concept in forming HCDSs. Available services are discovered in the network and choreographed to fit users' needs.

By following the SOA loose-coupling principle, services come from different sources, and are developed in different ways and, in general, without any coordination among developers. In this scenario, for a given system, uniformity of the adopted SOC technologies and middleware cannot be assumed in general. Thus, an integration solution at the middleware level is strongly required. Targeting the enterprise domain, Enterprise Application Integration (EAI) and Enterprise Service Bus (ESB) emerged as integration paradigms. An ESB can play the role of an intermediary among heterogeneous middleware and offered interaction paradigms through the integration of a set of reusable bridges. Another approach [25] to protocol interoperability relies on reflection, by dynamically plugging-in the most appropriate communication protocols according to the protocols sensed in the environment.

However, when engineering choreography-based service-oriented systems, where a large number of heterogeneous services are called to collaborate, it is unfeasible to assume a full knowledge of all the heterogeneity facets, e.g., all the interaction protocols run by these services. In fact, a choreography specification predicates on a set of abstract roles that, in the general case, are fulfilled only at run-time. In other words, the binding phase can take place only after concrete services have been discovered as suitable participants able to fulfill the required roles. Moreover, considering the future Internet[1] as the evolution of the current Internet, the service world is evolving at a very fast pace. For instance, up to now, a large amount of RESTful and/or WS* services can be discovered all over the Web (see ServiceFinder,[2] WebServiceList,[3] RemoteMethods,[4] WSCE,[5] ProgrammableWeb[6]).

In this scenario, it is likely to have several services providing similar functionalities through different interfaces and/or interaction protocols. This implies that unforeseen communication protocols will emerge, newly coming services will appear, and running choreographies are required to evolve. This prevents the selection of a specific middleware that, in the best case, can support all the bridges that solve interoperability among all the kind of services that are known at selection time. This means that the frontier between a choreography-based service-oriented application and its SOM cannot be a priori established once and for all, rather it varies as the choreography evolves.

Considering mobile devices with limited capabilities that access and provide services makes matters worse. Indeed, mobile computing environments are much more heterogeneous than conventional ones (see Sect. 3.1). This further restricts the possibility of selecting a middleware that provides seamless integration and interoperability through general purpose service-oriented interaction patterns (Fig. 2). The development of such a general purpose middleware is, per se, challenging since it should have built-in support for

[1] http://services.future-internet.eu/.

[2] http://www.service-finder.eu/.

[3] http://www.webservicelist.com/.

[4] http://www.remotemethods.com/.

[5] http://www.uoguelph.ca/ qmahmoud/WSCE/index.html.

[6] http://www.programmableweb.com.

as many mobile applications, old and new, as possible. On the other hand, mobile SOA applications need to rely on a run-time support for service access and provision, and will do so for long time in the future.

Therefore, a future engineering process is required to enable the development of adaptable and light-weight SOA-based applications together with SOM solutions capable to simultaneously evolve. When saying "capable to simultaneously evolve" we mean that both the application logic and the middleware support should be "upgraded or downgraded" to obtain the right balancing that fits the restrictions imposed by different mobile devices and permits to keep the evolution pace of the service world.

3.3 Cloud systems

The general promise of Cloud computing [38], as a more recent paradigm with respect to Grid computing, is to offer distributed computing infrastructure together with a set of technologies for elaborating and storing data. CPU power and ready-to-use software are provisioned by sharing and coordinating resources within a virtual organization. The technology behind cloud computing promises cheaper and flexible IT components, that may be rented instead of acquired upfront, and quickly scaled up and down according to the end-user's needs.

Mature virtualization technology is currently allowing the generalization of modern, more flexible service offerings ranging from virtualized hardware (Infrastructure as a Service—IaaS) such as Amazon EC2,[7] to application platforms (Platform as a Service—PaaS) such as SalesForce and Google Apps Engine, to complete applications such as Google Apps (Software as a Service—SaaS) [31]. More specifically, IaaS refers to very basic computing capability machines with operating systems and storage. PaaS is between IaaS and SaaS and refers to an environment where to build and run an application platform in the cloud using whatever prebuilt components, and interfaces are provided by that particular PaaS platform. SaaS refers to a software delivery model in which ready-to-use applications and related data are hosted in the cloud and made available over a network.

As it is clear from the discussion above, cloud systems constitute the most representative example of HCDSs. Promoting the philosophy of "*Everything as a Service*," cloud computing offers the modular and distributed virtualization of a complete system at different levels, with the (promised) high added value of paying only for what is needed when it is needed, by making the used resources vary in accordance with user needs. Whether a user decides to make use of SaaS, PaaS, or IaaS or any combination of them for a particular solution depends on a number of factors, including the availability of a SaaS application that directly suits the needs, the level of expertise in developing custom applications over PaaS, and the amount of required flexibility in many resources notably IaaS ones.

Since many years, SOC middleware are considered to be good candidates to support Grid and Cloud computing [26]. On the one hand, many Grid-oriented middleware have been proposed, e.g., OurGrid [14], InteGrade [33], to enable the execution of computationally-intensive applications on sets of highly distributed clusters of machines. On the other hand, Cloud computing as a more recent paradigm providing virtualization mechanisms specifically calls for a Cloud-oriented middleware for supporting more elastic and on-demand provision of remote networked resources at different levels, i.e., at SaaS, PaaS, or IaaS level. Even though at these three levels we can have a uniform representation of services, each level requires specialized means for managing services at run time, as well as, specialized features for supporting extra-functional crosscutting concerns.

In the following, we mention some middleware that support cloud computing at the IaaS, PaaS, and SaaS levels.

Cloud middleware at the IaaS level OpenNebula[8] (from the FP7 Reservoir project[9]) is an open-source platform for the management of virtualized data centers to build any type of IaaS cloud: private, public, virtual private, and hybrid. OpenNebula aims at providing an open, flexible, extensible, and comprehensive management layer to automate and orchestrate the operation of virtualized data centers by leveraging and integrating existing solutions. Eucalyptus[10] (which is used by Ubuntu for their cloud offering) enables the creation of IaaS private clouds, with no requirements for retooling the organization's existing IT infrastructure or need to introduce specialized hardware. Eucalyptus promotes a modern infrastructure virtualization software to create elastic pools that can be dynamically scaled up or down depending on applications workloads. Cloud.com[11] is an open source cloud computing platform for building and managing private and public cloud. It enables simple and cost effective deployment, management, and configuration of cloud computing environments, regardless of where they are deployed.

Cloud middleware at the PaaS level WSO2 Stratos[12] is an enterprise-grade and open PaaS, which provides the core

[7] http://aws.amazon.com/ec2.

[8] http://opennebula.org/.

[9] http://www.reservoir-fp7.eu/.

[10] http://www.eucalyptus.com/.

[11] http://www.cloud.com/.

[12] http://wso2.com/cloud/stratos/.

cloud services and essential building blocks required for developing SaaS and cloud applications. Oracle Fusion Middleware [31] identifies in the platform level of PaaS the right balance between flexibility and ease of use for the cloud customers. It proposes a foundation for realizing private cloud by delivering effective dynamic resourcing. Dynamic resourcing is complemented by user interaction technologies together with modularity, sharability, and composability, thus providing a powerful self-service platform of reusable components.

Cloud middleware at the SaaS level In [1], it is presented an architecture for achieving multitenancy at the SaaS level. Multitenancy allows a single application to emulate multiple application instances. This enables users to run services in a multitenant SOA framework and to build multitenant applications. This middleware is implemented on top of the WSO2 Carbon platform, already mentioned above when presenting cloud middleware at the PaaS level. CloudPointe[13] is a mobile SaaS middleware to enable mobile access to digital assets and shared documents on the cloud. CloudPointe solves data duplication, privacy, integrity, and security issues.

In line with the cloud computing philosophy, which distinguishes the three computing levels mentioned above, all the efforts toward the realization of cloud middleware adopt a separation-of-concerns approach. To this end, each middleware provides run-time support to the specific level it was born for. The perpetual engineering process model in Fig. 1 can be then instantiated at any level spanning from SaaS to IaaS. This calls for a new dimension of flexibility to be taken into account when engineering cloud-based HCDSs; this flexibility somehow subverts the three-levels vision of cloud computing, where middleware at higher levels mask details of lower levels. Thus, as for context-aware adaptable systems and SOA systems, this means that the frontier between the application and middleware cannot be defined once and for all. Even though the physical division imposed by the layered architecture of cloud computing is still valid, the future development process should provide a logical vision that crosscuts the three levels of middleware. This allows us to reach the best tradeoff by suitably selecting the most appropriated supporting features of the different levels in the cloud.

4 Conclusion and final remarks

In this paper, we consider different kind of HCDSs within three application domains, namely, context-aware adaptable systems, service-oriented systems, and cloud systems. Our thesis is that these systems require a futuristic dynamic development process that breaks the traditional boundaries between static and dynamic activities as well as, the frontier between the application and middleware layer.

This means that, for HCDSs a standard development process, where an application-domain-specific middleware is selected once and for all and employed [25], is unfeasible. The envisioned dynamic development process includes a perpetual phase in which the run-time-support features offered by the middleware-layer are selected and customized, assembled and evolved together with the application logic, the characteristics of the execution context, as well as, user needs. In other words, both the application logic and the middleware support should be capable to simultaneously evolve, meaning that they should be "upgraded or downgraded" to obtain the right balancing that fits the application requirements, the computing, and the networking environment, while satisfying the user expectations.

Acknowledgements This work is supported by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreements: number 257178 (project CHOReOS—Large Scale Choreographies for the Future Internet—www.choreos.eu), and number 231167 (project CONNECT—Emergent Connectors for Eternal Software Intensive Networked Systems—http://connect-forever.eu/).

[13] http://www.cloudxy.com.

References

1. Azeez A, Perera S, Gamage D, Linton R, Siriwardana P, Leelaratne D, Weerawarana S, Fremantle P (2010) Multi-tenant SOA middleware for cloud computing. In: Proceedings of CLOUD'2010, pp 458–465
2. Baldauf M, Dustdar S, Rosenberg F (2007) A survey on context-aware systems. Int J Ad Hoc Ubiquitous Comput 2(4):263–277
3. Baresi L, Ghezzi C (2010) The disappearing boundary between development-time and run-time. In: Proceedings of FoSER'10. ACM, New York, pp 17–22
4. Bencomo N, Blair GS (2009) Using architecture models to support the generation and operation of component-based adaptive systems. In: Software engineering for self-adaptive systems, pp 183–200
5. Blair GS, Coulson G, Andersen A, Blair L, Clarke M, Costa F, Duran-Limon H, Fitzpatrick T, Johnston L, Moreira R, Parlavantzas N, Saikoski K (2001) The design and implementation of open ORB 2. IEEE Distrib Syst Online 2:2001
6. Blair G, Bencomo N, France RB (2009) Models@run.time. Computer 42:22–27. http://doi.ieeecomputersociety.org/10.1109/MC.2009.326
7. Bucchiarone A, Pelliccione P, Vattani C, Runge O (2009) Self-Repairing systems modeling and verification using AGG. In: WICSA/ECSA'09, pp 181–190
8. Caporuscio M, Marco AD, Inverardi P (2007) Model-based system reconfiguration for dynamic performance management. J Syst Softw 80(4)
9. Caporuscio M, Raverdy PG, Issarny V (2011) UbiSOAP: a service oriented middleware for ubiquitous networking. IEEE Trans Serv Comput 99
10. Capra L, Emmerich W, Mascolo C (2003) Carisma: context-aware reflective middleware system for mobile applications. IEEE Trans Softw Eng 29:929–945

11. Cardellini V, Casalicchio E, Grassi V, Lo Presti F, Mirandola R (2009) Qos-driven runtime adaptation of service oriented architectures. In: ESEC/FSE '09. ACM, New York
12. wen Cheng S, cheng Huang A, Garlan D, Schmerl B, Steenkiste P (2004) Rainbow: architecture-based self-adaptation with reusable infrastructure. IEEE Comput 37:46–54
13. Cheng BHC, de Lemos R, Giese H, Inverardi P, Magee J (eds) (2009) Software engineering for self-adaptive systems (outcome of a Dagstuhl Seminar). LNCS, vol 5525. Springer, Berlin
14. Cirne W, Brasileiro F, Andrade N, Costa L, Andrade A, Novaes R, Mowbray M (2006) Labs of the world, unite!!! J Grid Comput 4
15. Cowan C, Black A, Krasic C, Pu C, Walpole J, Consel C, Volanschi E (1996) Specialization classes: an object framework for specialization. In: Proceedings of IWOOOS'96, Seattle, Washington
16. Epifani I, Ghezzi C, Mirandola R, Tamburrelli G (2009) Model evolution by run-time parameter adaptation. In: Proceedings of the 31st ICSE, pp 111–121
17. Eramo R, Malavolta I, Muccini H, Pelliccione P, Pierantonio A (2011) A model-driven approach to automate the propagation of changes among Architecture Description Languages. Softw Syst Model
18. Gamma E, Helm R, Johnson R, Vlissides JM (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading
19. Garlan D, Cheng SW, Huang AC, Schmerl B, Steenkiste P (2004) Rainbow: architecture-based self-adaptation with reusable infrastructure. Computer 37(10):46–54
20. Henricksen K, Indulska J (2005) Developing context-aware pervasive computing applications: models and approach. Pervasive Mob Comput 2:2005
21. Hirschfeld R, Costanza P, Nierstrasz O (2008) Context-oriented programming. J Object Technol 7(3):125–151
22. yi Hong J, ho Suh E, Kim SJ (2009) Context-aware systems: a literature review and classification. Expert Syst Appl 36(4): 8509–8522
23. Inverardi P (2007) Software of the future is the future of software. In: Trustworthy global computing. LNCS, vol 4661, pp 69–85
24. Inverardi P, Tivoli M (2009) Software engineering. In: The future of software: adaptation and dependability. Springer, Berlin, pp 1–31
25. Issarny V, Caporuscio M, Georgantas N (2007) A perspective on the future of middleware-based software engineering. In: Future of software engineering
26. Issarny V, Georgantas N, Hachem S, Zarras A, Vassiliadist P, Autili M, Gerosa M, Hamida A (2011) Service-oriented middleware for the future Internet: state of the art and research directions. J Internet Serv Appl 2:23–45
27. Keays R, Rakotonirainy A (2003) Context-oriented programming. In: Proceedings of MobiDe'03. ACM, New York
28. Kochan S (2003) Programming in Objective-C. Sams, Indianapolis
29. Liu H, Parashar M, Member S (2005) Accord: a programming framework for autonomic applications. IEEE Trans Syst Man Cybern 36:341–352
30. Peper C, Schneider D (2008) Component engineering for adaptive ad-hoc systems. In: SEAMS'08. ACM, New York
31. Piec M (2011) Platform-as-a-service private cloud with oracle fusion middleware. http://www.oracle.com/us/technologies/cloud/036500.pdf. Last ac: 28 Oct, 2011
32. Rouvoy R, Barone P, Ding Y, Eliassen F, Hallsteinsen SO, Lorenzo J, Mamelli A, Scholz U (2009) Music: middleware support for self-adaptation in ubiquitous and service-oriented environments. In: Software engineering for self-adaptive systems, pp 164–182
33. da Silva e Silva FJ, Kon F, Goldman A, Finger M, de Camargo RY, Filho FC, Costa FM (2010) Application execution management on the InteGrade opportunistic grid middleware. J Parallel Distrib Comput 70
34. Tanter É, Gybels K, Denker M, Bergel A (2006) Context-aware aspects. In: Software composition. LNCS, vol 4089, pp 227–242
35. Villazón A, Binder W, Ansaloni D, Moret P (2009) Advanced runtime adaptation for java. In: Proceedings of GPCE'09. ACM, New York, pp 85–94
36. Villazón A, Binder W, Ansaloni D, Moret P (2009) Hotwave: creating adaptive tools with dynamic aspect-oriented programming in java. In: Proceedings of GPCE'09. ACM, New York, pp 95–98
37. Volanschi EN, Consel C, Muller G, Cowan C (1997) Declarative specialization of object-oriented programs. In: OOPSLA, pp 286–300
38. Zhang Q, Cheng L, Boutaba R (2010) Cloud computing: state of the art and research challenges. J Int Serv Appl 1(1)